LA-UR-22-21007

Approved for public release; distribution is unlimited.

Title:	Profiling and Optimization
Author(s):	Li, Ying Wai
Intended for:	Simulating Physics using Efficient and Effective code Development (SPEED) program. This is a LANL internal lecture series for computational science application developers
Issued:	2022-02-07

Los Alamos
NATIONAL LABORATORY

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Profiling and Optimization

SPEED Lecture Series

Ying Wai Li (CCS-7)

February 1, 2022

I have a performance profile of my code. Now what?

- Look for the part that uses most computing time or memory transfer
- Observe the nature of tasks
  - DO loops/for loops?
  - How is the memory access pattern like?
  - I/O?
- Optimization techniques:
  - Make good use of compilers
  - Optimize memory access
  - Vectorization
  - Parallelization
  - I/O optimization

Important elements to look for

For serial codes:

- Hotspots
  Location (routines or lines) where most time is spent
- Loop structures
  Identify candidates for parallelization (threading and vectorization)
- Are there bottlenecks in the logical flow?
  e.g. C++ locks, mutex and futures

For parallel codes:

- MPI message passing
  Size of messages, where most communication is performed
- Load imbalance
- Overlapping of computation and communication

Do less work!

- Simplify expressions

$$x = Ace^{cy}$$

$$A = -\frac{1}{c}e^{cb}$$

$$x = -e^{c(b+y)}$$

```
b = 2.0
c = 3.0
A = -1.0/c * exp(c*b)
x = A*c*exp(c*y)
```

5 multiplications, 1 division, 2 exponentials

```
x = -exp(c*(b+y))
```

1 multiplications, 1 addition, 1 exponential

Do less work!

- Common subexpression elimination

```
x = cos(v)*(1+sin(u/2)) + sin(w)*(1-sin(u/2))
```

```
t = sin(u/2)
x = cos(v)*(1+t) + sin(w)*(1-t)
```

- Avoid expensive operations

```
y = x**2
```

```
y = x*x
```

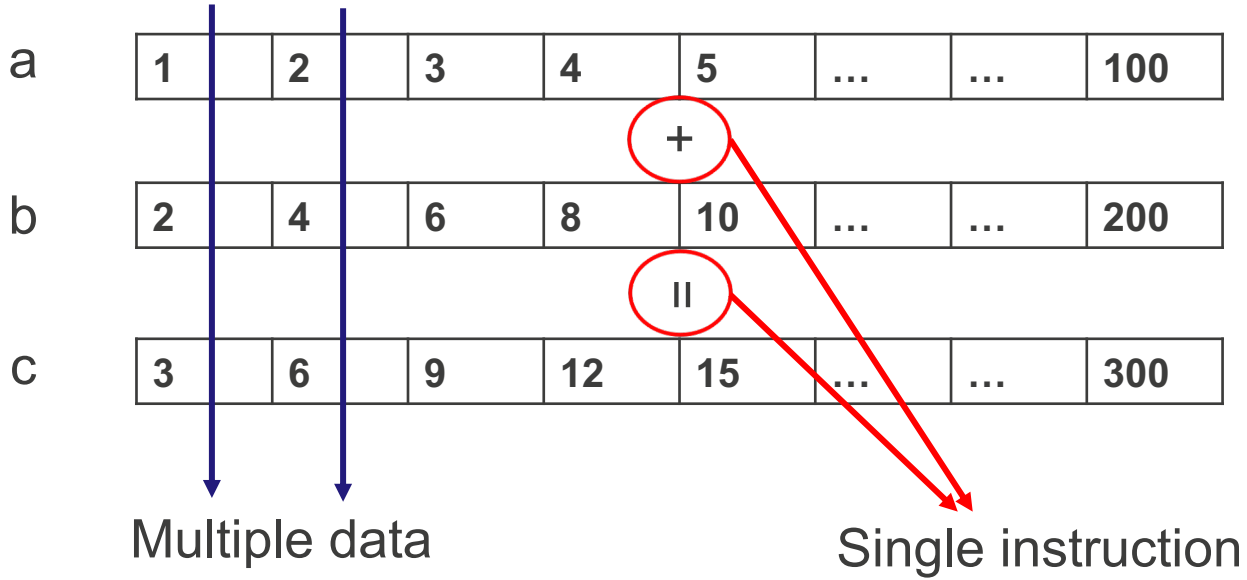
```
y = x/10
```

```
y = 0.1*x
```

Single Instruction Multiple Data (SIMD)

```
for (i=0; i<100; ++i)
   c[i] = a[i] + b[i];
```

```
DO i = 1, 100
   c[i] = a[i] + b[i]
END DO
```



Aside: Row-major order vs column-major order

Different ways to store multidimensional arrays in linear storage

- Row-major order: (C/C++, Python, Numpy)

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

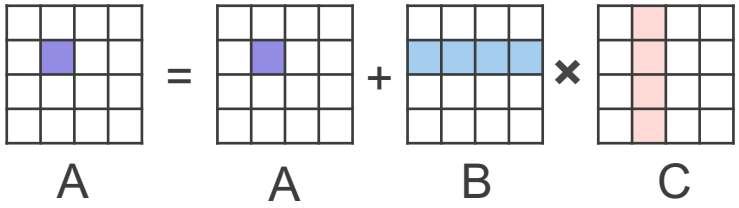
a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}	a_{31}	a_{32}	a_{33}

- Column-major order: (Fortran, MATLAB, R, Julia)

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}

Example: Matrix-matrix multiplication



Question 1:
Are these two programs the same?

Question 2:
Which one is better? Why?

Question 3:
How can we optimize them?

Fortran 90:

```
! Matrices A, B, and C initialized

DO k = 1, N
   DO j = 1, N
      DO i = 1, N
         A(i,j) = A(i,j) + B(i,k) * C(k,j)
      END DO
   END DO
END DO
```

C++:

```
// Matrices A, B, and C initialized

for (int k=0; k<N; ++k) {
   for (int j=0; j<N; ++j) {
      for (int i=0; i<N; ++i) {
         A[i][j] += B[i][k] * C[k][j];
      }
   }
}
```

Loop reordering

Useful when you want to:

- get contiguous memory access
- vectorize on one of the indices

How will you improve the previous C++ matrix multiplication code?

```
// Matrices A, B, and C initialized

for (int k=0; k<N; ++k) {
   for (int j=0; j<N; ++j) {
      for (int i=0; i<N; ++i) {
         A[i][j] += B[i][k] * C[k][j];
      }
   }
}
```

```
// Matrices A, B, and C initialized

for (int k=0; k<N; ++k) {
   for (int i=0; i<N; ++i) {
      for (int j=0; j<N; ++j) {
         A[i][j] += B[i][k] * C[k][j];
      }
   }
}
```

Loop unrolling

Note:

Computational intensity = $\frac{\text{floating point operations}}{\text{memory operations}}$

Useful when you want to:

- increase computational intensity within a loop
- give compiler more operations to shuffle around to get more overlaps

How will you improve the previous Fortran matrix multiplication code?

```
! Matrices A, B, and C initialized
DO k=1,N
   DO j=1,N
      DO i=1,N
         A(i,j)=A(i,j) + B(i,k)*C(k,j)
      END DO
   END DO
END DO
```

```
! Matrices A, B, and C initialized
DO k=1,N-3,4
   DO j=1,N
      DO i=1, N
         A(i,j)=A(i,j) + B(i,k)*C(k,j)
                       + B(i,k+1)*C(k+1,j)
                       + B(i,k+2)*C(k+2,j)
                       + B(i,k+3)*C(k+3,j)
      END DO
   END DO
END DO
DO kk=k,N
   DO j=1,N
      DO i=1,N
         A(i,j)=A(i,j) + B(i,kk)*C(kk,j)
      END DO
   END DO
END DO
```

Avoid conditionals/branches within loops

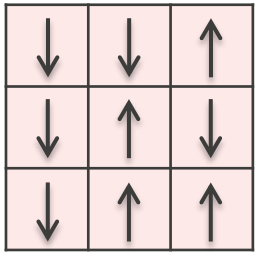
- Helps compiler vectorize the code
- For loop-independent if's:
  - remove (replace with intrinsics like `MAX`, `MIN`, `ABS`, or mathematical expressions)
  - pull them out from the loop

```
DO k=1,N
   DO j=1,N
      DO i=1,N
         IF (case==1) THEN
            A(i,j)=A(i,j) + B(i,k)*C(k,j)
         ELSE
            A(i,j)=A(i,j) + B(i,k)*D(k,j)
         END IF
      END DO
   END DO
END DO
```

```
IF (case==1) THEN
   DO k=1,N
      DO j=1,N
         DO i=1,N
            A(i,j)=A(i,j) + B(i,k)*C(k,j)
         END DO
      END DO
   END DO
ELSE
   DO k=1,N
      DO j=1,N
         DO i=1,N
            A(i,j)=A(i,j) + B(i,k)*D(k,j)
         END DO
      END DO
   END DO
END IF
```

Avoid conditionals/branches within loops

- For loop-dependent if's:
  - remove (replace with intrinsics like `MAX`, `MIN`, `ABS`, or mathematical expressions)
- Example: 2D Ising model with periodic boundary conditions



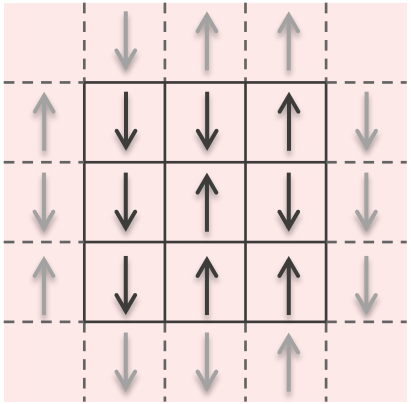
$$E = -J \sum_{<ij>} \sigma_i \sigma_j$$

```
int spin[Size][Size];
……
for (x=0; x<Size; ++x) {       // Energy calculation
   if (x!=0) xLeft = x-1;
   else xLeft = Size-1;
   for (y=0; y<Size; ++y) {
      if (y!=0) yBelow = y-1;
      else yBelow = Size-1;
      E += spin[x][y]*(spin[xLeft][y]+spin[x][yBelow]);
      M += spin[x][y];
   }
}
E *= -J;
```

How will you vectorize the energy calculation?

Avoid conditionals/branches within loops

- For loop-dependent if's:
  - remove (replace with intrinsics like `MAX`, `MIN`, `ABS`, or mathematical expressions)
- Example: 2D Ising model with periodic boundary conditions



$$E = -J \sum_{<ij>} \sigma_i \sigma_j$$

```
int spin[Size+2][Size+2];          // Create buffer halo space
for (x=0; x<Size; ++x) {           // Copy the spins to the halos
   spin[x][0] = spin[x][Size];
   spin[x][Size+1] = spin[x][1];
}
for (y=0; y<Size; ++y) {
   spin[0][y] = spin[Size][y];
   spin[Size+1][y] = spin[1][y];
}
for (x=1; x<Size+1; ++x) {         // Energy calculation
   if (x!=0) xLeft = x-1;
   else xLeft = Size-1;
   for (y=1; y<Size+1; ++y) {
      if (y!=0) yBelow = y-1;
      else yBelow = Size-1;
      E += spin[x][y]*(spin[x-1][y]+spin[x][y-1]);
      M += spin[x][y];
   }
}
E *= -J;
```

Loop splitting

- Useful when you want to:
  - remove non-vectorizable components from large computational loops
  - split non-tightly nested loops
- Caveat: does not always work because it increases memory movements

```
cmax = 0.0; imax = -1;
for (i=0; i<1000000; ++i){
   ctmp = a[i] + b[i];
   if (ctmp > cmax) {
      cmax = ctmp;
      imax = i;
   }
}
```

```
cmax = 0.0; imax = -1;
for (i=0; i<1000000; ++i)
   c[i] = a[i] + b[i];
for (i=0; i<1000000; ++i){
   if (c[i] > cmax) {
      cmax = c[i];
      imax = i;
   }
}
```

vectorizable

Loop splitting

- Useful when you want to:
  - remove non-vectorizable components from large computational loops
  - split non-tightly nested loops
- Caveat: does not always work because it increases memory movements

```
for (i=0; i<N; i++){
   a[i] = b[i]*2.0;
   for (j=0; j<M; j++){
      c[i,j] = d[j] + a[i];
   }
}
```

```
for (i=0; i<N; i++){
   a[i] = b[i]*2.0;
}
for (i=0; i<N; i++){
   for (j=0; j<M; j++){
      c[i,j] = d[j] + a[i];
   }
}
```

vectorizable

vectorizable

Loop fusion

- Useful when you want to:
  - get better cache utilization or reuse by minimizing loads and stores
  - increase computational intensity within a loop

```
for (i=0; i<N; ++i){
   for (j=0; j<M; ++j){
      a[i,j] = b[i] + c[j];
   }
}
for (i=0; i<N; ++i){
   for (j=0; j<M; ++j){
      d[i,j] = b[i] - c[j];
   }
}
```

```
for (i=0; i<N; ++i){
   for (j=0; j<M; ++j){
      a[i,j] = b[i] + c[j];
      d[i,j] = b[i] - c[j];
   }
}
```

Inline functions

- A good trick to optimize C++ applications that call small functions
- Eliminate function-call overhead
- Allow the compiler to resolve data dependencies
- How to use:
  1. Define the function as an inline function

```
float add(float a, float b){
  return a+b;
}
```

```
inline float add(float a, float b){
  return a+b;
}
```

  2. Use compiler flags
     Cray: `-hipa[n]. -hipa0` ignores all inlining directives; `-hipa5` most aggressive
     Intel: `-inline`
     GCC: `-finline-functions`
  3. Use compiler directives
     Cray: `#pragma inline_enable`

Make good use of compiler flags and directives

- Modern compilers do many of the optimization techniques
  e.g. loop reordering, unrolling, inlining, etc.
- Can be explicitly specified by the use of directives
- Some common optimization flags to use:
  - `-O0, -O1, -O2, -O3, -Ofast`
  - `-funroll-loops`

- Quick C++ benchmark: http://quick-bench.com/
  An online tool to examine and benchmark effects of different compilers and optimization levels
- Compiler explorer: https://gcc.godbolt.org/
  A website for exploring different compilers and how they generate assembly codes

Aside: How compilers optimize a code?

- To generate vector instructions, compilers:
  1. Identify a construct that performs a series of similar operations (typically a loop)
  2. Perform data dependency analysis
- To convert a series of operations into a vector operation, the operations have to work on a set of data that have been computed upfront

Some good practices:

1. Write codes in a way that hints/makes the compilers to vectorize
2. Avoid items that prevent vectorization by the compiler
   - Loop-carried dependencies
   - Indirect addressing
   - Excessive gathers/scatters/striding within a loop
   - Complex decision processes in a loop
3. Read documentations: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

```
DO i = 1, N
    a(ia(i)) = b(ib(i)) + c(ic(i))
END DO
```